

**UNIVERSIDAD TECNOLÓGICA DE SANTIAGO
(UTESA)**



PRESENTADO POR:

ERIK CRUZ 1-18-0759

PRESENTADO A:

IVAN MENDOZA

ASIGNATURA:

COMPILADORES

ASIGNACION:

COMPILADOR FINAL

**Santiago de los Caballeros
República Dominicana
Diciembre, 2023**

Introducción

Este proyecto trata sobre crear un programa especial llamado "compilador" que hace que programar sea más fácil para las personas que prefieren usar un lenguaje similar al español. La idea es que, en lugar de escribir código de programación de la manera convencional, puedan usar un estilo más parecido al español. Por ejemplo, si alguien quiere hacer un programa en Java, pero encuentra difícil la forma tradicional de programar, este compilador les permite escribir en un lenguaje que se parece más al español, y luego lo convierte automáticamente al código que la computadora entiende. Los objetivos del proyecto son diseñar las reglas de este nuevo lenguaje, crear herramientas para revisar que el código esté bien escrito, y generar el código final que la computadora pueda ejecutar. El proyecto se basa en entender cómo funcionan los compiladores, desde la traducción del lenguaje hasta la detección y corrección de errores en el código.

Capítulo 1: Descripción del Proyecto

1.1 Problema (Cual es el problema para nosotros tener que desarrollar este compilador)

El desarrollo de este compilador surge como respuesta a la necesidad de facilitar la programación para aquellos usuarios que prefieren expresarse en un lenguaje similar al español. La barrera del lenguaje convencional de programación puede resultar un obstáculo para aquellos que desean iniciarse en la programación, pero se sienten más cómodos con una sintaxis más cercana al español.

1.2 Planteamiento de la Solución (Propuesta de lo que su compilador)

La solución propuesta es un compilador capaz de traducir un lenguaje estructurado similar al español a código fuente en Java. Este compilador actúa como un puente entre el lenguaje natural y el lenguaje de programación, permitiendo a los usuarios expresar sus ideas de manera más intuitiva, fomentando así la participación en el ámbito de la programación.

1.3 Objetivos del Proyecto (Objetivos que quieren alcanzar con su compilador)

1.3.1 Objetivo General

Desarrollar un compilador eficiente y robusto que permita la traducción de un lenguaje estructurado similar al español a código Java, facilitando la programación para aquellos usuarios que no están familiarizados con la sintaxis convencional de los lenguajes de programación.

1.3.2 Objetivos Específicos

- **Diseñar la Gramática del Lenguaje Intermedio:** Definir las reglas gramaticales y la estructura del lenguaje intermedio que será traducido al código Java.
- **Implementar el Analizador Léxico y Sintáctico:** Desarrollar herramientas para analizar y validar la entrada del usuario, garantizando la correcta estructura del código fuente.
- **Generar Código Java:** Implementar un generador de código eficiente que traduzca el lenguaje intermedio a código Java.
- **Manejar Errores de Forma Amigable:** Proporcionar mensajes de error claros y comprensibles para facilitar la corrección de errores en el código fuente.

Capítulo 2: Marco Teórico

2.1 Introducción a los Compiladores

Un compilador es una herramienta fundamental en el campo de la informática que desempeña un papel crucial en la ejecución de programas de computadora. Su función principal es traducir el código fuente escrito en un lenguaje de programación de alto nivel a un código objeto ejecutable por la máquina. Este proceso permite a los programadores escribir código en un lenguaje más legible y comprensible, mientras que la máquina ejecuta instrucciones en un lenguaje más eficiente y cercano al hardware.

2.1.1 Definición de un compilador

Un compilador es un programa que toma como entrada un código fuente en un lenguaje de alto nivel y genera como salida un programa equivalente en un lenguaje de bajo nivel, generalmente el lenguaje de la máquina o código intermedio. Este proceso se realiza en varias fases, cada una encargada de una parte específica de la traducción.

2.1.2 Estructura de un Compilador

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico
- Generador de Código intermedio
- Generador de Código final
- Optimizador de Código
- Manejo de errores

2.2 Análisis Léxico

El análisis léxico es la primera fase del compilador que descompone el código fuente en tokens, unidades básicas como identificadores, palabras clave y operadores. Se realiza mediante un analizador léxico que escanea el código, reconociendo patrones y construyendo una lista de tokens. Estos tokens representan elementos clave del lenguaje y se utilizan en las fases posteriores del compilador para comprender la estructura del programa. Por ejemplo, en un código como "if (x > 5) { y = 10; }", los tokens incluirían palabras clave, operadores y símbolos, proporcionando una representación estructurada para el análisis sintáctico subsiguiente.

2.3 Análisis Sintáctico

El análisis sintáctico, verifica que el código fuente siga las reglas gramaticales del lenguaje. Utiliza la lista de tokens generada por el análisis léxico para construir un árbol de sintaxis abstracta que representa la estructura lógica del programa. Este árbol es esencial para entender la jerarquía de operaciones, las estructuras de control y otros aspectos fundamentales del código. Un ejemplo práctico sería la creación de un árbol para una estructura condicional "if-else" en un programa.

2.3.1 Análisis sintáctico o descendente

El análisis sintáctico descendente es un enfoque que parte desde el símbolo no terminal más alto y desciende hacia las hojas del árbol de sintaxis abstracta. Utiliza reglas gramaticales para reconocer la estructura del código, emplea una técnica llamada "análisis predictivo", y se beneficia al correlacionarse directamente con la gramática del lenguaje. Aunque puede ser más complejo de implementar, facilita la construcción de árboles de sintaxis abstracta.

2.4 Análisis Semántico

El análisis semántico es una fase clave en los compiladores que evalúa el significado del código fuente más allá de su estructura gramatical. Realiza tareas como la verificación de tipos, control de flujo semántico y resolución de nombres. Utiliza el árbol de sintaxis abstracta del análisis sintáctico y emite informes sobre errores semánticos. Por ejemplo, en la asignación de una cadena a una variable numérica, el análisis semántico identificaría y reportaría este error.

2.4.1 Traducción dirigida por la sintaxis

La traducción dirigida por la sintaxis en compiladores asocia acciones semánticas con las reglas gramaticales durante el análisis sintáctico. Cada regla gramatical tiene acciones asociadas que se ejecutan al reconocer la regla, permitiendo la generación de información semántica relevante. Este enfoque es esencial para construir información semántica necesaria para etapas posteriores del compilador, como la generación de código intermedio.

2.4.2 Atributos

Los atributos son valores asociados a elementos gramaticales durante el análisis sintáctico. Pueden ser sintetizados o heredados, y se utilizan para capturar información semántica, como tipos de datos o alcance de variables. Estos atributos son fundamentales para la transferencia de información semántica a lo largo del proceso de análisis sintáctico, enriqueciendo el árbol de sintaxis abstracta con detalles relevantes para etapas posteriores del compilador.

2.4.3 Notaciones para asociar reglas semánticas

Las notaciones para asociar reglas semánticas en compiladores, como la Definición Dirigida por la Sintaxis (SDD), SDD con Atributos y SASD, proporcionan enfoques formales para especificar acciones semánticas durante el análisis sintáctico. Estas notaciones permiten definir y gestionar acciones semánticas, junto con atributos, en el contexto de las reglas gramaticales. Facilitan una representación clara y estructurada de la semántica del programa, mejorando la comprensión y mantenimiento del compilador.

2.4.3.1 Definición dirigida por la sintaxis

La Definición Dirigida por la Sintaxis (SDD) en compiladores asocia acciones semánticas directamente con las producciones gramaticales durante el análisis sintáctico. Cada regla gramatical incluye acciones que especifican cómo interpretar esa parte del código, facilitando la generación eficiente de información semántica durante la construcción

del árbol de sintaxis abstracta. Esta técnica es esencial para el proceso de compilación, mejorando la eficiencia y claridad del compilador.

2.4.3.2 Esquema de traducción

El esquema de traducción define cómo traducir construcciones gramaticales de un lenguaje fuente a su equivalente en un lenguaje destino, incorporando reglas semánticas. Utiliza producciones gramaticales para asociar acciones semánticas, explicando de manera sistemática la transformación de sintaxis y semántica entre ambos lenguajes. Este enfoque es esencial para guiar la traducción de un compilador y asegurar la coherencia entre el código fuente y el código destino.

2.5 Generación de Código Intermedio

La Generación de Código Intermedio consiste en crear una representación abstracta y portable del programa fuente. Se realiza mediante técnicas como la Traducción Dirigida por la Sintaxis y el Esquema de Traducción, generando código de tres direcciones, árboles de expresiones o código de máquina virtual. Esta etapa simplifica la generación de código final y permite implementar optimizaciones globales antes de la generación específica para la máquina objetivo. La representación intermedia sirve como puente entre el análisis sintáctico/semántico y la fase final de generación de código.

2.6 Generación de Código Final

La Generación de Código Final es la fase donde se traduce el código intermedio en código ejecutable específico para la arquitectura de la máquina objetivo. Implica asignar registros, generar instrucciones y aplicar optimizaciones locales y globales. El resultado es un programa optimizado y directamente ejecutable, aprovechando las características de la máquina objetivo para mejorar la eficiencia. Ejemplos incluyen la producción de código ensamblador o código máquina.

2.7 Tablas de Símbolos y Tipos

Las Tablas de Símbolos almacenan información sobre identificadores (variables, funciones) y las Tablas de Tipos sobre los tipos de datos. Ambas se actualizan durante el análisis semántico y son cruciales para garantizar consistencia y coherencia en el programa compilado. Estas tablas facilitan la gestión de información relacionada con variables, funciones y tipos de datos durante todo el proceso de compilación.

2.8 Entorno de Ejecución

El Entorno de Ejecución es el contexto donde un programa compilado se ejecuta. Incluye componentes como el área de memoria, la pila de llamadas y registros de programa. Gestiona variables, funciones, llamadas y retornos, asignación de memoria y flujo de ejecución. Durante la generación de código final, se considera la estructura del entorno para asignar registros y gestionar la pila de llamadas de manera eficiente. Es esencial para la correcta ejecución de un programa compilado.

2.9 Optimización de Código

La Optimización de Código busca mejorar la eficiencia del código generado sin cambiar su funcionalidad. Se realiza en etapas locales y globales, aplicando técnicas como eliminación de código muerto, propagación de constantes, optimización de bucles y

reordenamiento de instrucciones. El objetivo es reducir el tiempo de ejecución y el uso de memoria. Estas optimizaciones se aplican después de la generación de código intermedio y antes de la generación de código final, buscando un equilibrio entre complejidad y beneficio.

2.9 Manejo de Errores

El Manejo de Errores en compiladores es crucial para detectar y corregir problemas en el código fuente durante la compilación. Involucra la identificación de errores sintácticos, semánticos, léxicos y de tipo en diversas etapas del proceso, con estrategias de recuperación como la inserción o eliminación de tokens. Además, proporciona mensajes claros al usuario para facilitar la corrección del código y busca continuar la compilación de manera robusta después de detectar errores.

2.10 Herramientas de Construcción de Compiladores

Las Herramientas de Construcción de Compiladores son programas que facilitan el desarrollo de compiladores. Incluyen generadores de analizadores léxicos y sintácticos como Lex, Yacc y ANTLR, frameworks como LLVM y GCC, así como ambientes integrados de desarrollo como Eclipse y IntelliJ IDEA. Además, existen herramientas específicas para lenguajes como javac y PyInstaller, generadores de código intermedio como CodeDOM y Gforth, y herramientas de análisis estático como Coverity. Make y CMake automatizan el proceso de compilación, y lenguajes específicos de dominio como Racket permiten construir compiladores para dominios particulares. Estas herramientas se eligen según las necesidades y el lenguaje objetivo del compilador en desarrollo.

Capítulo 3: Desarrollo

3.1 Analizador Léxico

3.1.1 Autómata del análisis Léxico

El autómata del análisis léxico describe cómo se reconocen los diferentes tokens en el código fuente.

Palabras reservadas:

```
r.put(k: "INICIO", v: 0);
r.put(k: "FINAL", v: 0);
r.put(k: "WORD", v: 0);
r.put(k: "ALFA", v: 0);
r.put(k: "NUM", v: 0);
r.put(k: "DNUM", v: 0);
r.put(k: "BOOL", v: 0);
r.put(k: "LNUM", v: 0);
r.put(k: "LEER", v: 0);
r.put(k: "ESCRIBIR", v: 0);
r.put(k: "CUANDO", v: 0);
r.put(k: "SI", v: 0);
r.put(k: "IS", v: 0);
r.put(k: "DESDE", v: 0);
r.put(k: "PASO", v: 0);
r.put(k: "HASTA", v: 0);
r.put(k: "FDESDE", v: 0);
r.put(k: "FCUANDO", v: 0);
r.put(k: "FSI", v: 0);
```

Operadores

```
op.put(k: "/", v: 0);
op.put(k: "*", v: 0);
op.put(k: "+", v: 0);
op.put(k: "-", v: 0);
op.put(k: "=", v: 0);
op.put(k: "^", v: 0);
op.put(k: "<", v: 0);
op.put(k: ">", v: 0);
op.put(k: "||", v: 0);
op.put(k: "&&", v: 0);
```

Delimitadores

```
deli.put(k: "#", v: 0);
deli.put(k: ";", v: 0);
deli.put(k: "{", v: 0);
deli.put(k: "}", v: 0);
deli.put(k: ")", v: 0);
deli.put(k: ",", v: 0);
deli.put(k: "(", v: 0);
```

3.1.2 Tabla de Símbolos

Ejemplo con programa de factorial de un numero

Tabla de Símbolos		
Token	Cantidad	Tipo
FCUANDO	1	Palabra ...
NUM	1	Palabra ...
FINAL	1	Palabra ...
CUANDO	1	Palabra ...
ESCRIBIR	4	Palabra ...
SI	3	Palabra ...
INICIO	1	Palabra ...
FSI	3	Palabra ...
*	1	Operador
+	5	Operador
<	2	Operador
=	11	Operador
>	1	Operador
#	10	Delimita...
(8	Delimita...
)	8	Delimita...
;	15	Delimita...
{	5	Delimita...
,	3	Delimita...
}	5	Delimita...
Factorial	1	Identific...
numero	8	Identific...

3.1.3 Implementación del análisis léxico

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // Creación de estructuras de datos para contar tokens  
    HashMap<String, Integer> r = new HashMap<>();  
    HashMap<String, Integer> op = new HashMap<>();  
    HashMap<String, Integer> id = new HashMap<>();  
    HashMap<String, Integer> deli = new HashMap<>();  
    HashMap<String, Integer> num = new HashMap<>();  
    LinkedList<String> texto = new LinkedList<>();  
  
    // Inicialización de contadores para palabras reservadas  
    r.put("INICIO", 0);  
    // (se omite el resto por brevedad)  
  
    // Configuración del modelo de la tabla  
    DefaultTableModel model = new DefaultTableModel();  
    model.setColumnIdentifiers(new Object[]{"Token", "Cantidad", "Tipo"});  
  
    // Tokenización del texto ingresado  
    StringTokenizer st = new StringTokenizer(txtATexto1.getText(), "{}();\n\"+=*/><|&&# \n\t\", true);  
    String token, text = \"\";  
    while (st.hasMoreTokens()) {  
        token = st.nextToken();  
        if (!\"\".equals(token) && !\"\\n\".equals(token) && !\"\\t\".equals(token)) {  
            // Actualización de contadores según el tipo de token  
            if (r.containsKey(token)) {  
                r.put(token, r.get(token) + 1);  
            } else {  
                // (se omite el resto por brevedad)  
            }  
        }  
    }  
  
    // Iteración sobre las palabras reservadas y actualización del modelo de la tabla  
    Iterator<String> itr = r.keySet().iterator();  
    while (itr.hasNext()) {  
        token = itr.next();  
        if (r.get(token) > 0) {  
            model.addRow(new Object[]{token, r.get(token), \"Palabra Reservada\"});  
        }  
    }  
  
    // (se omite el resto por brevedad)  
  
    // Establecimiento del modelo de la tabla  
    tabla.setModel(model);  
}
```

Este código Java realiza las siguientes acciones paso a paso:

1) Creación de Estructuras de Datos:

- Se crean HashMaps (r, op, id, deli, num) para contar diferentes tipos de tokens.
- Se crea una LinkedList (texto) para almacenar texto entre delimitadores #.

2) Configuración del Modelo de la Tabla:

Se crea un modelo de tabla (DefaultTableModel) con columnas "Token", "Cantidad" y "Tipo".

3) Tokenización del Texto:

Se utiliza un StringTokenizer para dividir el texto ingresado en tokens, usando varios delimitadores.

4) Iteración sobre Tokens y Conteo:

- Se itera sobre cada token obtenido del StringTokenizer.
- Se verifica y cuenta la frecuencia de palabras reservadas (r).

5) Iteración sobre Palabras Reservadas y Actualización del Modelo:

Se itera sobre las palabras reservadas y se actualiza el modelo de la tabla con la información relevante.

6) Establecimiento del Modelo de la Tabla:

Se establece el modelo de la tabla (tabla) con el modelo creado, lo que actualiza la interfaz gráfica con los resultados del análisis léxico.

3.2 Analizador Sintáctico

3.2.1 Definición de Gramáticas libres del Contexto

3.2.2 Tipo de Analizador Sintáctico (LL, LR o ninguno)

no implementa explícitamente un analizador sintáctico (LL, LR, etc.). este realiza una tokenización manual y verifica patrones específicos en el texto de entrada, en lugar de depender de un generador de analizadores sintácticos o una biblioteca de combinación de analizadores formales.

3.2.3 Autómata del análisis Sintáctico

Este no presenta un autómata finito determinista (DFA) o no determinista (NFA) explícito para el análisis sintáctico. Sin embargo, sugiere un enfoque de análisis sintáctico descendente manual mediante la tokenización de la entrada y la verificación de patrones mediante expresiones regulares.

Cada no-terminal se asocia con funciones que intentan reconocer la estructura correspondiente. Aunque el código no utiliza un generador de analizadores sintácticos, el enfoque general implica la construcción de un autómata basado en la gramática del lenguaje, la tokenización de la entrada y la implementación de funciones recursivas para el análisis sintáctico.

3.2.3 Implementación del análisis Sintáctico

Expresiones Regulares:

El código define varias expresiones regulares para representar patrones específicos del lenguaje, como identificadores, números, cadenas, operadores, estructuras de control, etc.

Estas expresiones regulares se utilizan para realizar coincidencias y verificar la sintaxis de las partes del programa.

Tokenización con StringTokenizer:

Se utiliza para separar el programa en bloques definidos por los delimitadores `; y {}`, así como para dividir líneas de código.

```
String token = "";
```

Se utiliza un bucle para recorrer los tokens y realizar acciones según el tipo de token encontrado.

```
while (st.hasMoreTokens()) {
    token = st.nextToken();
    // ...

    // Lógica para analizar el tipo de token y realizar
    acciones correspondientes.

    // ...
}
```

Se realizan comprobaciones semánticas, como asegurarse de que las variables estén declaradas antes de su uso y validar las asignaciones según el tipo de datos.

```
if (ENT.contains(tipo) || DEC.contains(tipo) ||  
TEXT.contains(tipo) || TAKE.contains(tipo)) {  
    // Manejo de error: Variable duplicada.
```

```
// ...  
}
```

Construcción de Estructuras de Control:

Los contadores (start, when, it) se utilizan para seguir el flujo de las estructuras de control como bucles y condicionales.

Se realizan comprobaciones para asegurarse de que las estructuras de control estén correctamente balanceadas.

```
if (token.matches(start2)) {  
    start++;  
}  
  
if (token.matches(start3)) {  
    start--;  
}  
  
// ... otras verificaciones de estructuras de control ...
```

3.3 Reglas del Lenguaje

1) Variables y Tipos de Datos:

El código reconoce variables con nombres que cumplen la expresión regular `id = "[a-zA-Z][\w]*"` (letras seguidas de letras o números).

Se definen tres tipos de datos: NUM, DNUM (números decimales), y WORD (cadenas de texto).

2) Operaciones Aritméticas:

Se permiten operaciones aritméticas básicas (+, -, *, /) entre variables de tipo numérico (id, num, dec).

3) Sentencias de Entrada/Salida:

Se utilizan las sentencias LEER y ESCRIBIR para la entrada y salida de datos, respectivamente.

4) Estructuras de Control:

Se definen estructuras de control como SI (if), FDESDE (fin de bucle for), FCUANDO (fin de condición), etc.

5) Declaración de Variables:

Se definen variables con sus tipos (NUM, DNUM, WORD) utilizando sentencias como `NUM x;`, `DNUM y;`, etc.

6) Ciclos y Condiciones:

Se tienen estructuras para bucles (DESDE, FDESDE) y condiciones (CUANDO, FCUANDO, SI, FSI).

7) Inicio y Fin del Programa:

Se utiliza INICIO y FINAL para delimitar el bloque principal del programa.

3.4 Generador de Código Intermedio

Este traduce el lenguaje escrito a un lenguaje intermedio. El código utiliza expresiones regulares para identificar patrones específicos en el código fuente original y realiza la correspondiente traducción a instrucciones del lenguaje BASIC

Algunas de las operaciones que realiza el método incluyen la traducción de instrucciones de entrada y salida, la declaración de variables, la estructura de control de flujo (como condicionales y ciclos), entre otros. Es importante tener en cuenta que el código podría necesitar adaptaciones adicionales para manejar todas las posibles construcciones del lenguaje intermedio.

3.5 Manejo de Errores

Se realiza un análisis léxico y una validación semántica del un código fuente. se utiliza expresiones regulares para identificar patrones en el código, se emplean listas enlazadas para el seguimiento de variables y errores, y analiza estructuras de control como ciclos y condiciones. La validación semántica incluye la verificación de la declaración adecuada de variables y la generación de mensajes de error en caso de incongruencias. Además, el programa habilita o deshabilita un botón de traducción según la presencia de errores.

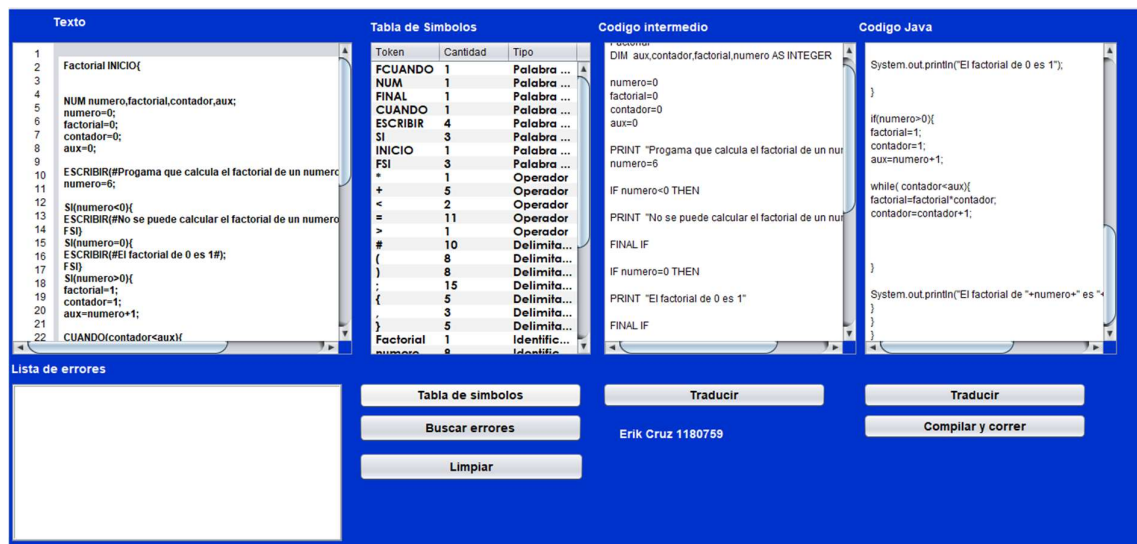
3.7 Creación del Proyecto

3.7.1 Pasos para crear el proyecto

- 1) Se necesita tener una versión de java 21 o mayor instalada
- 2) Ejecutar .exe
- 3) Escribir instrucciones según las reglas definidas

3.7.2 Explicación del funcionamiento del proyecto

Pantalla de inicio:



Panel de texto: Aquí se escribirán las instrucciones en base a las reglas ya definidas

Texto

1

```
Factorial INICIO{

    NUM numero,factorial,contador,aux;
    numero=0;
    factorial=0;
    contador=0;
    aux=0;

    ESCRIBIR(#Progama que calcula el factorial de un numero
    numero=5;

    SI(numero<0){
    ESCRIBIR(#No se puede calcular el factorial de un numero
    FSI}
    SI(numero=0){
    ESCRIBIR(#El factorial de 0 es 1#);
    FSI}
    SI(numero>0){
    factorial=1;
    contador=1;
    aux=numero+1;

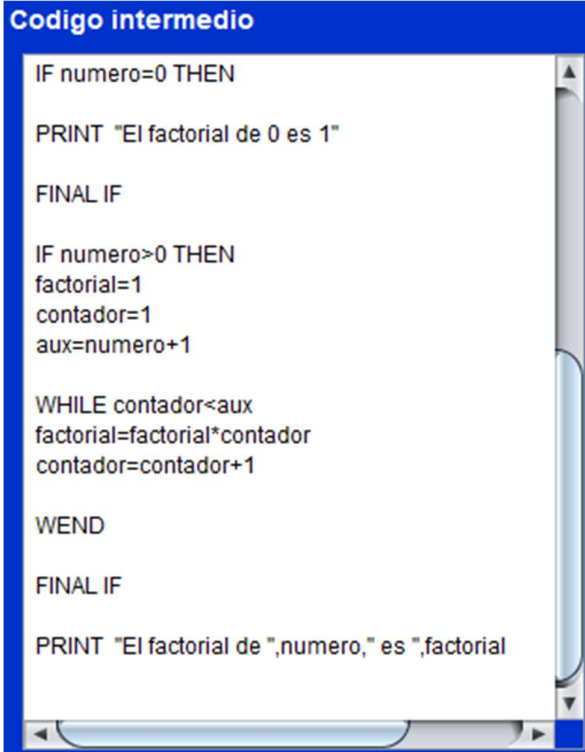
    CUANDO(contador<aux){
```

Tabla de símbolos: esta almacena detalles clave sobre identificadores, simplificando la gestión durante la compilación.

Tabla de Símbolos

Token	Cantidad	Tipo
FCUANDO	1	Palabra ...
NUM	1	Palabra ...
FINAL	1	Palabra ...
CUANDO	1	Palabra ...
ESCRIBIR	4	Palabra ...
SI	3	Palabra ...
INICIO	1	Palabra ...
FSI	3	Palabra ...
*	1	Operador
+	5	Operador
<	2	Operador
=	11	Operador
>	1	Operador
#	10	Delimita...
(8	Delimita...
)	8	Delimita...
;	15	Delimita...
{	5	Delimita...
,	3	Delimita...
}	5	Delimita...
Factorial	1	Identific...
numero	9	Identific...

Codigo intermedio: Aquí se mostrara el código intermedio generado

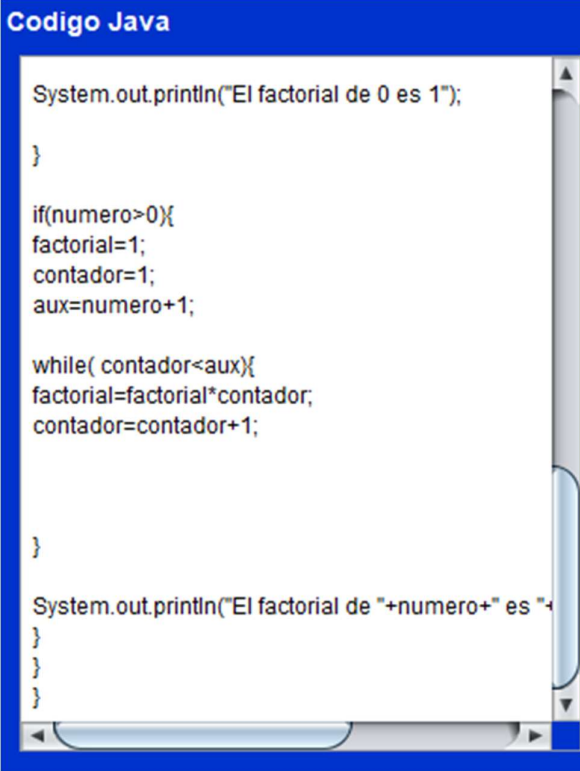


```
IF numero=0 THEN  
  
PRINT "El factorial de 0 es 1"  
  
FINAL IF  
  
IF numero>0 THEN  
factorial=1  
contador=1  
aux=numero+1  
  
WHILE contador<aux  
factorial=factorial*contador  
contador=contador+1  
  
WEND  
  
FINAL IF  
  
PRINT "El factorial de ",numero," es ",factorial
```

Lista de errores: Aquí se mostraran los errores



Codigo java: Aquí se vera el código traducido a java



```
System.out.println("El factorial de 0 es 1");

}

if(numero>0){
factorial=1;
contador=1;
aux=numero+1;

while( contador<aux){
factorial=factorial*contador;
contador=contador+1;

}

System.out.println("El factorial de "+numero+" es "+factorial);
}
}
```

Botones: Lista de botones para hacer las acciones correspondientes



https://github.com/kelok3rik/COMPILADOR_DEF

CONCLUSION

En conclusión, este proyecto de compilador se presenta como una solución innovadora para superar las barreras del lenguaje convencional de programación, facilitando la entrada al mundo de la programación a aquellos usuarios que se sienten más cómodos expresándose en un lenguaje estructurado similar al español. A través de un enfoque manual en el análisis léxico y sintáctico, junto con la generación de código intermedio, el compilador busca traducir este lenguaje estructurado a código Java.

Los objetivos del proyecto, que incluyen el diseño de la gramática del lenguaje intermedio, la implementación de analizadores léxicos y sintácticos, la generación de código Java y el manejo amigable de errores, se alinean con la visión general del desarrollo de compiladores presentada en el marco teórico. Se abordan aspectos cruciales como las fases del compilador, el análisis semántico, la generación de código final y la optimización, asegurando una comprensión profunda de los principios subyacentes.

La implementación del analizador léxico y sintáctico, así como las reglas del lenguaje, demuestran un enfoque detallado en la validación y estructuración del código fuente, utilizando expresiones regulares y estructuras de control. El manejo de errores, a través de un análisis semántico y mensajes claros, mejora la experiencia del usuario al facilitar la detección y corrección de posibles incongruencias.